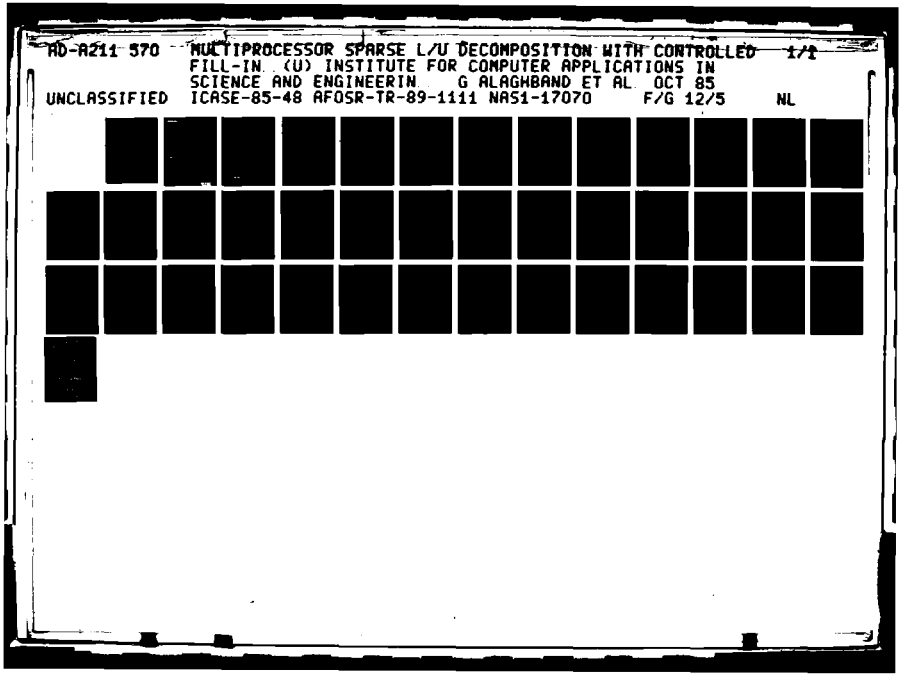
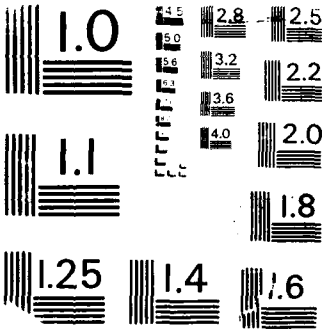

AD-A211 570 MULTIPROCESSOR SPARSE L/U DECOMPOSITION WITH CONTROLLED 1/1
FILL-IN. (U) INSTITUTE FOR COMPUTER APPLICATIONS IN
SCIENCE AND ENGINEERIN G ALAGHBAND ET AL. OCT 85
UNCLASSIFIED ICASE-85-48 AFOSR-TR-89-1111 NAS1-17070 F/G 12/5 NL


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6

(2)

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

DTIC
ELECTE
AUG 18 1989
B

AD-A211 570

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
	5. MONITORING ORGANIZATION REPORT NUMBER(S) AFOSR-TR- 89-1111
6a. NAME OF PERFORMING ORGANIZATION NASA Langley Res. Ctr.	6b. OFFICE SYMBOL (If applicable)
6c. ADDRESS (City, State, and ZIP Code) Inst for Comp Appl in Sci and Engin Hampton, VA 23665	7a. NAME OF MONITORING ORGANIZATION Air Force Office of Scientific Research
	7b. ADDRESS (City, State, and ZIP Code) Building 410 Bolling AFB, DC 20332-6448
8a. NAME OF FUNDING/SPONSORING ORGANIZATION AFOSR	8b. OFFICE SYMBOL (If applicable) NM
8c. ADDRESS (City, State, and ZIP Code) Building 410 Bolling AFB, DC 20332-6448	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER AFOSR-85-0189
	10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO	WORK UNIT ACCESSION NO.
61102F	2304	A3	

11. TITLE (Include Security Classification)
Multiprocessor Sparse L/U Decomposition with Controlled Fill-In

12. PERSONAL AUTHOR(S)
Gita Alaghband and Henry F. Jordan

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM 6-1-85 TO 8-31-87		

16. SUPPLEMENTARY NOTATION
[redacted]

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This effort supported the research activities of 20 researchers during their visit to ICASE, as a result, 10 papers have appeared on issues related to parallel computation including such titles as "Reordering computations for parallel execution", "Multiprocessor L/U decomposition with controlled fill-in, and "Analysis of a parallelized nonlinear elliptic boundary value problems solver with applications to reacting flows".

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS	UNCLASSIFIED

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code)	22c. OFFICE SYMBOL
Lt. Col David Nelson	(202) 767-5026	NM

DD Form 1473, JUN 86 — Previous editions are obsolete. — SECURITY CLASSIFICATION OF THIS PAGE

89 8 10 008

UNCLASSIFIED

NASA Contractor Report 178016

ICASE REPORT NO. 85-48

ICASE

MULTIPROCESSOR SPARSE L/U DECOMPOSITION WITH CONTROLLED FILL-IN

Gita Alaghband

Harry F. Jordan

Contract No. NAS1-17070, Grant No. AFOSR 85-0189

October 1985

INSTITUTE FOR COMPUTER APPLICATIONS IN SCIENCE AND ENGINEERING
NASA Langley Research Center, Hampton, Virginia 23665

Operated by the Universities Space Research Association

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

Multiprocessor Sparse L/U Decomposition with Controlled Fill-in

Gita Alaghband
and
Harry F. Jordan
Department of Electrical and Computer Engineering
University of Colorado

Abstract

During L/U decomposition of a sparse matrix, it is possible to perform computation on many diagonal elements simultaneously. Pivots that can be processed in parallel are related by a compatibility relation and are grouped in a compatible set. The collection of all maximal compatibles yields different maximum sized sets of pivots that can be processed in parallel. Generation of the maximal compatibles is based on the information obtained from an incompatible table. This table provides information about pairs of incompatible pivots. In this paper, generation of the maximal compatibles of pivot elements for a class of small sparse matrices is studied first. The algorithm involves a binary tree search and has a complexity exponential in the order of the matrix. Different strategies for selection of a set of compatible pivots based on the Markowitz criterion are investigated. The competing issues of parallelism and fill-in generation are studied and results are provided. A technique for obtaining an ordered compatible set directly from the ordered incompatible table is given. This technique generates a set of compatible pivots with the property of generating few fills. A new hueristic algorithm is then proposed that combines the idea of an ordered compatible set with a limited binary tree search to generate several sets of compatible pivots in linear time. Finally, an elimination set to reduce the matrix is selected. Parameters are suggested to obtain a balance between parallelism and fill-ins. Results of applying the proposed algorithms on several large application matrices are presented and analyzed.

---

†Research was supported in part by NASA Contract No. NAS1-17070 and by the Air Force Office of Scientific Research under Grant No. AFOSR 85-[illegible] while the authors were in residence at ICASE, NASA Langly Research Center, Hampton, VA 23665. 0189

Introduction

Solution of a linear system of equations is required in many application programs. One such area is the VLSI circuit simulation programs. Every computer-aided circuit analysis program includes a routine that solves a system of sparse linear equations. If implicit integration is used, at every time step one must solve a system of nonlinear equations (usually by Newton iteration). At every iteration a system of linear equations must be solved. Depending on the integration method, the number of times that a sparse system of linear equations needs to be solved may be large. If it is possible to reduce the solution time for the sparse system, the total circuit analysis time would be significantly reduced. One method for solving such a system is the factorization of the matrix into lower and upper triangular matrices followed by forward and back substitutions.

One promising area for advances in solution technique is the use of parallel computers and parallel algorithms. Our previous work on parallelizing the MA28 [1] sparse matrix package for the HEP [2] multiprocessor suggests that sufficient parallelism is not obtainable in sparse L/U decomposition without processing multiple pivots in parallel [3]. Parallel pivoting strategies have been investigated by Calahan [4] and more recently by Wing and Haung [5], [6], Jess and Kees [7] and Peters [8]. Although the number of operations possible in parallel may be large in a very sparse system, exploitation of all the available parallelism may significantly increase the generation of fill-ins (zero element of the matrix becoming nonzero as a result of elimination). Since fill-in increases the total computation work, it is important to keep the number generated under control. The purpose of this work is to study sparse L/U decomposition on a multiprocessor by means of an algorithm which exploits parallel pivots and keeps fill-in low. The class of sparse systems guiding the study will be those arising from the simulation of VLSI circuits using a program such as SPICE [9].

Wing and Haung in [5] represent the triangulation process by a directed graph where the vertices represent a divide or update operation (operations required for performing the triangulation), and the edges determine the precedence relation of the operations to be executed. By assigning level numbers to the directed graph, they identify all operations on the same level to be done in parallel. They use a weighted combination of fill-in cost and depth of computation in a heuristic to determine a nearly optimal pivot sequence. While Wing and Haung identify all the operations that can be done in parallel, we will identify all pivots that can be processed in parallel at each step. An issue that has not been discussed in the literature is that in a sparse matrix there are usually different sets of possible pivot candidates for each step, and the sizes of these sets may well vary. It seems important to study these possibilities and the effect of parallel pivoting on application matrices. Algorithms identifying parallel pivot candidates are complex, so it will be of value to come up with such algorithms only if the amount of parallelism in circuit domain matrices is large enough to justify the computation required to identify it.

In this paper, we assume a shared-memory, MIMD model for our parallel computation, in which the total memory address space is accessible uniformly to all parallel units (processes or individual processors). This computational model should provide synchronization mechanisms to allow multiple memory updates . If multiple updates are aimed at the same memory cell, the penalty paid is a short delay in access time. Based on this computational model, the first half of this paper is devoted to study the amount of parallelism that exists in application matrices. This is carried out by producing all possible sets of pivot candidates which can be processed in parallel at each step for a number of small matrices. Observations are then made on different strategies for choosing one of the sets produced at each step, and hence the generation of fill-ins and possible parallel pivoting steps. The complete and detailed analysis of this study leads us into the second half of the paper, where we describe a fast heuristic algorithm to produce a set of acceptable parallel pivot candidates for reducing the matrix at each step. Issues involved in balancing parallel work and fill-in generation are discussed and verified through simulated results.

Parallel Pivot Candidates

The triangulation method used here as mentioned above will be sparse L/U decomposition. For simplicity, we only consider the diagonal elements of the matrix as pivot candidates. Note that pivoting usually refers to unsymmetric permutations of the matrix for swapping an off-diagonal matrix element with a diagonal element. In this paper, we are only considering symmetric permutations of the matrix. Even though we are not pivoting in the above sense, the terms pivot and pivoting are used throughout the paper to refer to the diagonal element used to reduce the matrix at a given step and a symmetric permutation respectively.

In a sparse matrix, two pivots a_{ii} and a_{jj} can be processed in parallel if a_{ij} and a_{ji} are both zero. In other words, during elimination, row j is not involved in the elimination process taking place for pivot a_{ii}, and row i is not involved in the process for a_{jj}. This statement can only be true if we provide correct synchronizations for simultaneous update during the elimination with parallel pivot candidates:

1. During elimination, when processing pivots $a_{ii}, a_{jj}, \ldots$ in parallel, it is possible that an element of a nonpivot row needs to be updated by all or some of the parallel processes handling pivots i,j,... for the current step. In order for each process to obtain a completely updated value, as a result of a previous update, the update operation must be done asynchronously by parallel processes. On the other hand, the order in which parallel processes update an element is of no importance (except for round off errors).
2. During elimination, when processing pivots $a_{ii}, a_{jj}, \ldots$ in parallel, it is possible that a fill-in is generated in position (m,n). It is also possible that more than one process tries to generate a fill-in in the same position (m,n). The position (m,n) for the fill-in must be created once by one process only, and other processes will update its value as in 1.

If two pivots a_{ii} and a_{jj} can be processed in parallel, and if a_{jj} and a_{kk} can also be processed in parallel, then a_{ii},a_{jj}, and a_{kk} cannot necessarily be processed in parallel. The relation between parallel pivot candidates is *reflexive* and *symmetric*, but not *transitive*, and is thus a *compatibility* relation. Two pivots related in this way will simply be said to be *compatible* in what follows. A consequence of the nontransitivity of the compatibility relation is that it classifies the elements of a set into nondisjoint subsets, so that all members of a subset are compatible. These subsets are called compatibility classes. Thus, in order to come up with all possible sets of pivots that can be processed in parallel and are of maximum size, we need to find all maximal compatibles. A maximal compatible is a compatible that is not included in any larger compatible.

To clarify the discussion, we define a boolean matrix B for each sparse matrix A, such that:

$$b_{ij} = 1 \quad iff \quad a_{ij} \neq 0$$
$$b_{ij} = 0 \quad otherwise$$

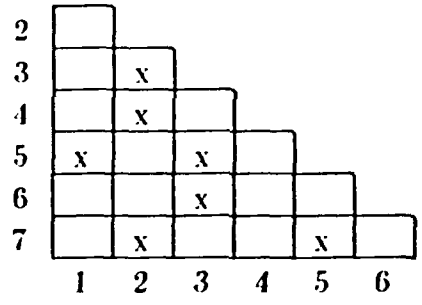
where b_{ij} and a_{ij} denote elements of B and A respectively.

Several approaches for constructing the set of maximal compatibles exist, and they are all based on construction of an incompatible table [10]. The incompatible table specifies pairs of incompatible elements. Assume pivots are taken from the diagonal elements of the sparse matrix and are numbered 1 through n corresponding to diagonal elements of rows 1 through n. Now we could represent the incompatible table as a table consisting of (n-1) columns, where each column i has (n-i) elements. Columns of the table correspond to pivot elements of the matrix. Column one of the table, corresponding to pivot number one, is set to the bit vector resulting from oring row and column one of the matrix B and keeping the last (n-i) elements. The same process is repeated for pivot 2 (column 2 of the table), for the submatrix obtained from the original matrix with row and column one eliminated. For every column of the table that is completely constructed, the corresponding row/column of the matrix is eliminated. The process is repeated for all pivots in order. It is important to note that the incompatible table is constructed for a given ordering of the sparse matrix. Thus, there are $n!$ different incompatible tables for $n!$ possible diagonal orderings of an n by n sparse matrix. In what follows, we represent the incompatible table as an array of dimension n, say *imptbl(n)*, with elements of the array being sets of at most n elements each. Each set corresponds to a column of the table. As an illustrative example, the incompatible table for the matrix A1 of Fig. 1.1.a is given in Fig. 1.1.b.

The maximal compatibles are found by combining the pivot-pairs from the incompatible table into larger groups with compatible elements. Several systematic approaches for extracting the maximal compatibles have been suggested, and they all use an exhaustive search routine. The one approach that seems to be more suitable for programming on a digital computer is one that assumes initially that all pivot candidates can be grouped into one set. Then

	1	2	3	4	5	6	7
1	x						
2		x		x			
3		x	x		x	x	
4				x			
5	x		x		x		
6						x	
7		x			x		x

Matrix A1
Fig. 1.1.a

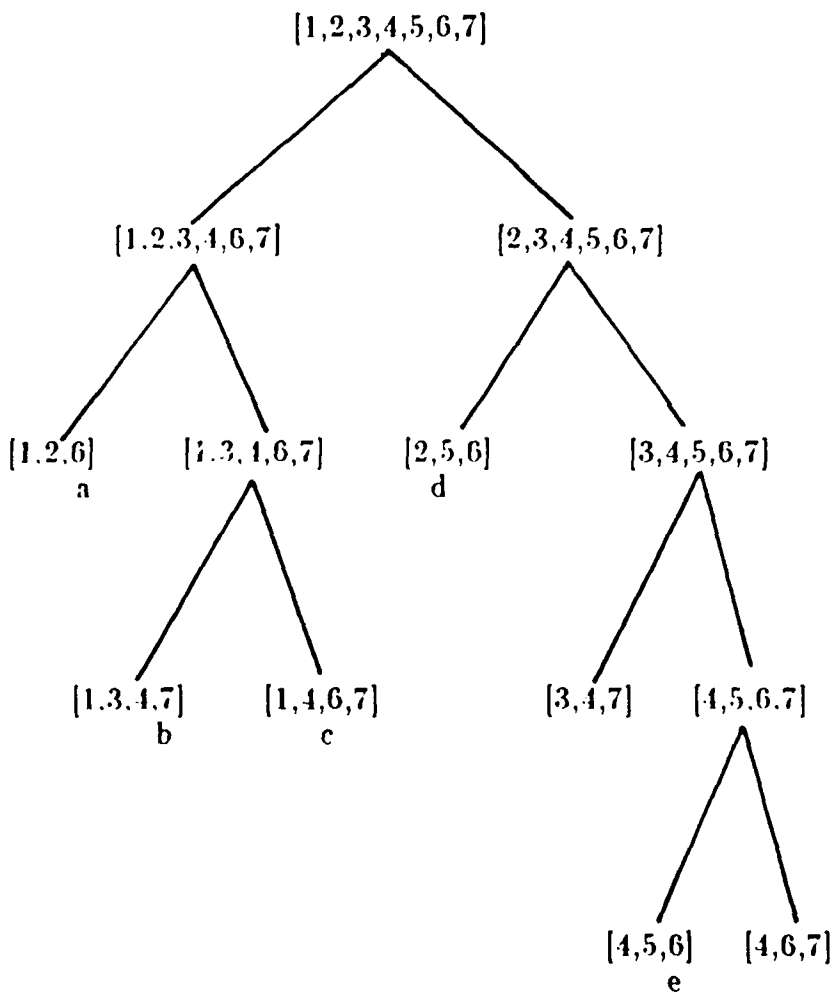


Incompatible Table
Fig. 1.1.b

the information from the incompatible table is used for contradictions and splitting the groups where necessary. This procedure involves searching a binary tree. Initially, it is assumed that all pivots are compatible. They are grouped in one set consisting of all pivot elements. This set will be at the root of a binary tree, level zero. Next, the set of pivots incompatible with pivot number one, obtained from the incompatible table, is used to split the set at the root into a left and a right set, constituting level one. The left set consists of all elements of its parent set at level zero, except those incompatible with pivot one. The right set consists of the same elements as the starting set (parent set), except pivot one itself. At the next step, the incompatible information for pivot number two, is used to break each set at level one into a left and right set for level 2. Furthermore, since the matrix is sparse, some of the sets at a given level will not split into smaller sets for some pivoting elements, but they may still consist of incompatible elements and will split for some later pivots. Consequently, the binary tree corresponding to this search will not always be a dense tree. This process is repeated until no

more splitting of the sets is possible. The leaf sets are then checked and every set included in a larger leaf set is eliminated. The remaining sets constitute all possible maximal compatibles. Note that the length of a path from the root to a leaf could be at most n.

The above process is shown for the example matrix of Fig. 1.1 in Fig 1.2. Initially, pivots number 1 through 7 are grouped together as the starting set. Column one of the incompatible table indicates that pivot 5 is incompatible with pivot one. Thus the starting set is split into two sets (1,2,3,4,6,7) and (2,3,4,5,6,7). At the next level, these two sets are broken into four sets, each using the incompatibility information for pivot number two from the table. This process is continued until no more splits are possible. At the end, the



Binary Tree Search to Obtain
the Set of Maximal Compatibles

Fig 1.2

extra sets (3,4,7) and (4,6,7) which are included in the maximal sets (b) and (c) respectively, are eliminated. The remaining five sets are the maximal compatibles.

A high level description of the above procedure is given below:

```
procedure MAXCOMP(sset,i)
Assumptions:
- pivot candidates are numbered from 1 to n.
- initially sset consists of all pivots in the matrix and
  i is the first pivot.
        while i<n do
        begin
        (*split sset into left and right sets*)
          lset = sset - imptbl[i]
          rset = sset - [i]
          if (lset not a compatible set) then
              maxcomp(lset,i+1)
          if(rset not a compatible set) then
              maxcomp(rset,i+1)
        end
```

In the above procedure, many branches do not need to be continued to the completion of the search, since they are included in other subtrees. Moreover, as will be described later, we only need to produce compatible sets of maximum size. Thus, there are many branches in this tree that could be trimmed to limit the amount of search. Even including these features, this algorithm has exponential complexity, and only serves to obtain information about sparse matrices.

To study the issues discussed earlier, a PASCAL program was written to perform symbolic L/U decomposition on a sparse matrix. Our objective was to study the effects of parallel pivoting so the program performs the decomposition to the last parallel step and does not continue if parallel pivot candidates are not available. The structure of the program is outlined below:

```
program PIVOTSET
- Read in input matrix and construct matrix structure.
-Construct all maximal compatibles.
  -if parallel pivoting is not possible go to stop
    -Pick a set of compatible pivots to be processed
       in parallel.
    -Permute the matrix according to the parallel pivots for this step.
    -reduce the matrix and insert the resultant fill-ins.
-Repeat.
-Stop.
```

Analysis Performed

In general, in matrices arising from circuits there are many different sets of compatible pivots of equal maximum size. Depending on how a set is chosen to reduce the matrix at each step, we obtain a different behavior in generation of fill-in elements, and as a result, different possibilities for continuing parallel pivoting in the next steps. The issues of generation of fill-ins and parallelism in pivoting have been studied. We used different strategies to select a set of compatible pivots and then obtained statistical information from some circuit matrices generated from the SPICE circuit simulation program.

The Markowitz criterion [11] is well known for minimizing the generation of fill-ins in sparse matrices in sequential programming. It is based on the fact that at step k, the maximum number of fill-ins generated by choosing a_{ij} as pivot is $(r_i-1)(c_j-1)$. Here r_i-1 is the number of nonzero elements other than a_{ij} in the $i-th$ row of the reduced matrix, and c_j-1 is the number of nonzero elements other than a_{ij} in column j of the reduced matrix. Markowitz selects as pivot element at step k, the element which minimizes $(r_i-1)(c_j-1)$. The product $(r_i-1)(c_j-1)$ is the Markowitz number of element a_{ij}. In what follows, we use the Markowitz idea as a basis for the selection of a compatible pivot set.

In our first analysis we compare two different strategies for choosing a set of compatible pivots among all maximal compatibles. In both cases we consider only the sets of maximum size. The first strategy (called Markowitz sum) chooses that set among all sets of maximum size in which the sum of the Markowitz numbers of all its elements is minimum. The problem here is that some of the pivots in the set chosen for reducing the matrix may generate fill-ins in the same positions, and thus we overestimate the Markowitz count for a purely sequential case. As an alternative, a second strategy is employed (called Ored Markowitz). Here, using the boolean matrix B corresponding to the sparse matrix under consideration, we count number of nonzeros in a vector that is the result of ORing rows of pivot candidates in the set and multiply this number by the number of non-zeros in a vector resulting from ORing columns of the potential pivots.

Comparison of the above strategies on our test cases shows that the first method is almost always superior. Our results show that, in general, by minimizing the Markowitz sum we always get fewer fill-ins generated and often more rows are reduced in parallel steps. This study has shown that the amount of parallelism in circuit matrices is quite high but that the generation of fill-in terms is also quite high in most cases when compared to the sequential runs on the same matrices. The number of potential pivots to be processed in parallel at each step seems to be so high that we could process fewer pivots in parallel in a step without limiting the parallel work considerably. An experiment to study this possibility is performed by picking the maximum sized set with minimum Markowitz sum as was explained above. This set is then used to reduce the matrix, with the following analysis performed on the set of compatible pivots.

-Discard the pivot with maximum Markowitz count and determine number of fill-ins that would be generated as a result.
-Repeat the above procedure until no more pivots can be discarded from the set, either because the set size is too small or because all Markowitz sums are zero.

Although the above analysis of reducing the size of the set of compatible pivots was done for each step, the actual elimination and fill for a step was done using the maximal compatible with lowest Markowitz sum. This analysis is repeated at each parallel step and the results show that it is possible to decrease the generation of fill-ins at this step significantly by reducing the amount of parallel work slightly. In fact, discarding only one compatible pivot results in a decrease of at least about one third in the number of fill-ins that would be generated otherwise.

We performed this analysis over all generated sets of compatible pivots also. In this experiment, we chose maximum sized set with minimum Markowitz sum and used it for reducing the matrix as described below:

- for all sets of maximum size do
  find the pivot with maximum Markowitz count and
  remove from the set.
- find the set of maximum size and minimum Markowitz sum and determine number of fill-ins that would be generated from the processing of this set.
- repeat the above process.

Similar results were obtained by applying the above two procedures to our test matrices. Hence, we will use the first method for the next phase. That is, the next analysis is performed on the set of maximum size and minimum Markowitz sum.

Even though the above experiment shows we can always generate fewer fill-ins at a step by avoiding the maximum possible parallelism, it does not indicate that this will not delay the generation of fill-ins to later steps. In our next experiment, we choose the maximum sized set with minimum Markowitz sum, but this time we discard the pivot with maximum Markowitz count from the set and use the resulting set for elimination and fill generation. We will also repeat the previous analysis by reducing the set size and determining number of resulting fill-ins. This work confirms our previous result that by discarding some of the parallel pivot candidates according to their high Markowitz count we decrease the total generation of fill-ins.

Results of Complete Analyses

A set of circuits to be simulated by the SPICE circuit simulation program is available as a benchmark to test SPICE. We used these circuits as input to SPICE and generated their corresponding matrices. These matrices are used as test cases for analysis purposes. The first circuit is a simple differential pair and generates a 16 by 16 matrix with 57 nonzeros. The matrices are of small sizes and the size range is between 12 by 12 to 21 by 21. The complexity of our algorithm to generate all possible maximal sets of compatible pivots would not allow us to test larger matrices, but the generated

information produces valuable statistics about parallelism and circuit matrices. An algorithm with tolerable complexity to produce a set of compatible pivots will involve heuristics; therefore, it will not give total information about the matrix.

The results of comparison of Markowitz sum and Ored Markowitz strategies are summarized in Table 1.1 (tables are provided in appendix A at the end of this paper). The first column describes the circuit, the order of the matrix, and number of nonzeros. The second column indicates the parallel pivoting step. Columns 3 to 5 correspond to the Markowitz sum strategy described earlier, and columns 6 through 8 correspond to Ored Markowitz. The first column for each algorithm is the size of the maximum set of pivots obtained at a step, the second column is the minimum operation count obtained for such a set, and the last column specifies the number of fill-ins that are generated as the result of processing the indicated set. Column 9 indicates the total number of maximal compatibles generated at each step. The last two columns are information generated by the SPICE program about the amount of fill-in generated and the percentage of the matrix which is zero.

As can be seen from the table, in every case the second strategy resulted in equal or more fill-ins and equal or fewer parallel steps with fewer number of rows reduced. This indicates that the Markowitz sum is a better heuristic for selecting the set of pivots among many sets. This can be observed from the 16 by 16 matrix of the differential pair circuit. In the first step, with sets of size six, Markowitz sum generated 6 fill-ins while Ored Markowitz generated 8. The pivot set chosen by the Markowitz sum generated fewer fill-ins than the Ored Markowitz algorithm, and, as can be seen, the Ored Markowitz resulted in twice as many fill-ins as the Markowitz sum and fewer pivots were processed in parallel (14 for Ored Markowitz and 15 for Markowitz sum). The same behavior resulted from the ECL compatible SCMITT trigger circuit which produced an 18 by 18 matrix. The number of fill-ins at step 2 of parallel triangulation is 10 for Ored Markowitz and only 4 for the Markowitz sum with none being generated in the next steps. Ored Markowitz generated 4 more fill-ins at step 3 and was not able to find any more parallel pivot candidates, but the first strategy continued to do one more parallel step. Of course, there are cases where both strategies produced similar or close results, as can be seen from the table. The table also indicates that, in parallel runs, generation of fill-ins is much higher than in sequential runs of the SPICE program. At the same time it can be seen that the matrices generally do not become dense rapidly, and parallel pivot candidates are available to almost the very last steps of the triangulation process.

The result of our next analysis is shown in Table 1.2. At each step, a set of maximum size and minimum Markowitz sum is selected to reduce the matrix. Furthermore, from this set we repeatedly remove a pivot with maximum Markowitz count and compute the number of fill-ins that would be generated if this set were used to reduce the matrix. As can be seen from the table, in every case it is possible to reduce number of fill-ins significantly by reducing the amount of parallelism slightly. For example, for the 16 by 16

matrix with 57 nonzeros, we can see that if we reduce the number of compatible pivots from 6 to 4 by removing the two pivots with highest Markowitz count from the set, we can prevent generation of more fill-ins. Also in the last 24 by 24 matrix with 158 nonzeros, we can reduce the number of generated fill-ins by a factor of 2 (from 40 to 20), if we discard two pivots in step one in the same fashion. This is a general result that can be observed from the table for all cases and all parallel steps.

In the next experiment we confirm that it is possible to reduce the total generation of fill-ins, as opposed to just at each step , by using fewer than the maximum number of compatible pivots. In every case we have been able to reduce the total number of fill-ins by some fraction (at least about one third), compared to the case where maximum parallelism was employed . These results are summarized in Table 1.3. Here we chose to discard a pivot from the maximal compatible set according to its highest Markowitz count. If the maximal set would not generate any fill-ins, because of a zero Markowitz sum, we did not discard any pivots from the set. The total number of fill-ins generated for the first matrix (16 by 16) is 2 which is one third of the amount generated with our first experiment (6). This number was reduced from 40 to 26 for the case of the 24 by 24 matrix with 151 nonzeros. In this case the number of parallel steps was increased from 5 to 6, but the total number of rows that could be reduced in these steps remained constant. In fact, in most cases, the number of parallel steps is increased, but the total number of pivots that could be processed in these steps does not change much (no change is greater than one addition or reduction in the number of reduced rows).

Generation of Compatible Sets from the Incompatible Table

It is clear that in large sparse circuit matrices the number of possible pivots to be processed at each step will be much higher than our small example matrices, and therefore, it will be possible to obtain enough parallel work by just considering a sub-maximal set of compatible pivots at each step. The algorithm described involves a complete binary tree search and has exponential complexity in the order, n, of the sparse matrix. In order to come up with a good heuristic, we need to relax the requirement of finding the maximal set of compatible pivots with minimum Markowitz sum. As a conclusion from the above analysis, we will have to reduce the size of the set to decrease the generation of fill-ins. Keeping these problems in mind, an acceptable set would be one which has a large number of pivot candidates for parallel processing and a low enough Markowitz sum. We now need to look for a procedure which tends to produce a number of compatible sets of reasonably large size and low Markowitz sum. Having generated such sets, we can then choose the best candidate among these compatible sets using the same criteria as before. In what follows, we will describe different issues which will lead us to a good heuristic algorithm and a set of parameters to be used in trading off between fill-in generation and the size of the set of parallel pivot candidates.

So far, the information from the incompatible table has been used to construct the maximal compatible sets of pivots in a complete binary tree search algorithm. A more careful analysis of the incompatible table could provide a set of compatible pivots without the need for searching the tree. As we know, this table gives information about the incompatible pairs of pivots. In other words, by looking at column i of the table corresponding to pivot i, we obtain all pivot numbers j>i where pivot j is incompatible with pivot i, for a given ordering of the matrix. Note that we are assuming pivots are taken from the diagonal of the matrix and they are numbered 1 through n corresponding to rows 1 through n of the matrix. Consequently, if column i of the table is null, then the corresponding pivot number i is compatible with every pivot whose corresponding column lies to the right of column i. Hence, by scanning the incompatible table, we can find a set of compatible pivots whose corresponding columns in the table are null. Clearly, pivots with such a property are compatible and can be grouped in a compatible set. Using the representation of the incompatible table described earlier, the above procedure can be formulated as:

```
scan imptbl from right to left
  for each column i of imptbl do
  , if ( imptbl_i is empty) then
    (*add the corresponding pivot to the set of compatibles*)
    compset = compset + [i]
```

where *compset* is the set of compatible pivots whose corresponding columns in the table are null. Now if there exists a pivot k such that the set of pivots incompatible with it in column k of the table, is disjoint from the set of already constructed compatible pivots in *compset* , then k is compatible with every pivot in *compset* . Therefore, we can expand *compset* by adding k to it. The above procedure can now be completely described as:

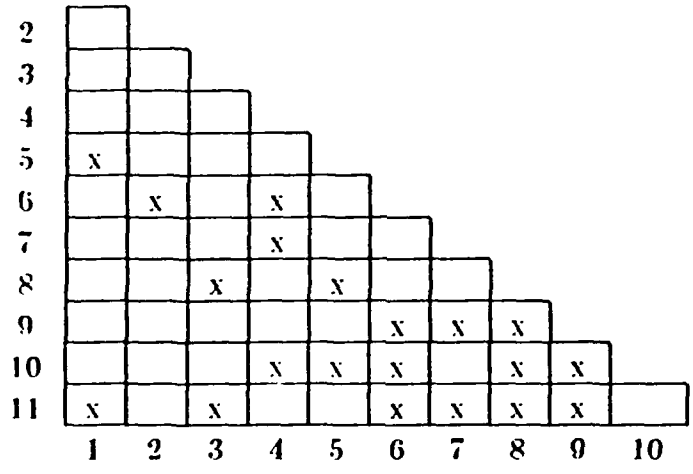
```
scan imptbl from right to left
  for each column i of imptbl do
    begin
    if ( imptbl_i ∩ compset is empty) then
    (*add [i] to the set of compatibles*)
    compset = compset + [i]
    else
    delete row i of imptbl
    end
```

The compatible set, *compset* , produced by this procedure, will be referred to as an ordered compatible set from now on, since it is obtained by imposing a specific ordering on the diagonal elements of the matrix to get the incompatible table. As an example, the incompatible table of matrix A2 in Fig. 2.1.a is given in Fig. 2.1.b. The compatible set corresponding to the null columns of the table consists of pivots 10 and 11. This set consists of 2, 10, and 11 after the above expansion.

As was explained previously, our strategy for selecting a compatible set among all possible compatible sets of equal maximum size was to select the

	1	2	3	4	5	6	7	8	9	10	11
1	x										
2		x									
3			x					x			x
4				x		x					
5	x				x					x	
6		x				x					
7				x			x		x		x
8			x		x			x	x		
9						x	x		x		x
10				x		x		x	x	x	
11	x					x		x	x		x

Matrix A2
Fig. 2.1.a



Incompatible Table
Null columns: (10,11)
Compset: (2,10,11)
Fig. 2.1.b

one with minimum Markowitz sum. That is, to select the set in which the sum of Markowitz numbers of the pivots in its set is minimum. If we consider the set of compatible pivots constructed above directly from the incompatible table, we see that it consists of pivots 2, 10, and 11, which in turn have Markowitz numbers 0, 4, and 12. In general, we would like to have a compatible set consisting of pivots with as low Markowitz numbers as possible. It is also clear that pivots with low Markowitz numbers generally have fewer incompatibilities. Moreover, by looking at the incompatible table of Fig. 2.1.b, we see the compatible pivots 10, 11 are obtained from the right end portion of the

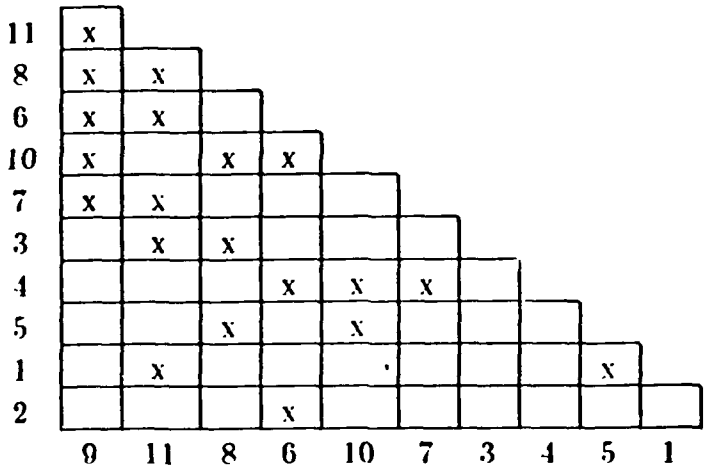
table. This is usually the case, since as we construct columns of the incompatible table, we are left with a smaller submatrix to work with. Thus, after completing each column, we have fewer incompatibles left for the construction of the next column. These observations lead us to use a different ordering in which the first column of the incompatible table has the maximum number of incompatibles and as we work our way to the right end of the table, the number of incompatibles will decrease to the minimum. Such an ordering implies the resulting incompatible table will have more null columns clustered at the right end. So the ordered compatible set that can be constructed from the ordered table will be of a larger size and smaller Markowitz sum than the results of the above procedure. As a result of these arguments, we sort the pivots in order of decreasing Markowitz numbers. Using this new ordering, we can construct a new incompatible table with the first column corresponding to the pivot with highest Markowitz number and the last column corresponding to the pivot with lowest Markowitz number. As an example, the Markowitz numbers and the new ordering of the pivots are shown in Fib. 2.2.a for matrix A2 of Fig. 2.1.a. The corresponding ordered incompatible table is given in Fig. 2.2.b. It can be seen from Fig. 2.2.b that the collection of pivots corresponding to null columns of the table gives a compatible set of size 4 and Markowitz sum 4 consisting of pivots 1, 2, 3, and 4. This is in comparison with set of size 2 and Markowitz sum 16 generated from the unordered incompatible table of Fig. 2.1.b. After expanding this set, we produce a compatible set of size 5 and Markowitz sum 16 consisting of pivots 1, 2, 3, 4, and 9.

Limited Binary Search Tree

In this section, we will combine the idea of an ordered compatible set with the tree search algorithm described earlier to obtain a limited tree

Pivot	Markowitz Number	Order
1	0	9
2	0	11
3	2	8
4	2	6
5	2	10
6	4	7
7	3	3
8	9	4
9	12	5
10	4	1
11	12	2

Fig. 2.2.a



Ordered Incompatible Table
Null columns: (1,2,3,4)
Compset: (1,2,3,4,9)
Fig. 2.2.b

search algorithm which produces an acceptable set of compatible pivots for *reducing the matrix*. Given a set of all pivot elements, we can now directly produce a set of compatible pivots from the ordered incompatible table. This ordered compatible set is obtained for the initial starting set at the root of the binary search tree. A child set in the tree is a subset of its parent set. In this context, every set at any given point in the tree has fewer pivots than the root set. Such a set could be considered as a starting set itself. Provided we could produce the correct incompatible table for this set, we could generate its corresponding ordered compatible set directly from the new table.

The incompatible table for a given starting set, S_i, is the original table with those rows and columns corresponding to the pivots absent from S_i eliminated. If we let S be the initial set of all pivot candidates in the sparse matrix and S_i be an arbitrary starting set in the tree , then the procedure to obtain the ordered compatible set for S_i , $compset_i$, from an updated and ordered incompatible table can be represented as:

```
1.    compset_i = empty
2.    less = S - S_i
3.    for j = n down to 1 do
4.      begin
5.      if ( j ∈ S_i ) then
6.        begin
7.        tempset = impltbl_j - less
8.        tempset = tempset ∩ compset_i
9.        if ( tempset = empty ) then
10.         compset_i = compset_i + [j]
11.       end
12.     end
```

where *less* is the set of pivots absent from S_i. Line 5 allows only those columns of the incompatible table whose corresponding pivot j is in S_i to be tested for the compatibility relation. Set *less* is used in line 7 to eliminate rows corresponding to the absent pivots in S_i. $compset_i$ holds the current set of compatible pivots. A check for a new pivot being compatible with those already in $compset_i$ is made in line 9.

It is now possible to produce an ordered compatible set for any set at any arbitrary point in the tree directly from the incompatible table. Given a starting set, our method of producing an ordered compatible set tends to generate a large set of low Markowitz sum. Thus, we can produce a number of ordered compatible sets for many starting sets at different points in the tree and choose the best candidate among them to reduce the matrix. The following theorem will eliminate of some of the redundant work.

Theorem

All ordered compatible sets derived from the starting sets in the binary search tree with level L-1 or less are included in the ordered compatible sets generated from the sets at level L of the tree. (i.e., it is only necessary to generate ordered compatible sets for starting sets at level L to cover those at level l<L.)

Proof

Let S be the initial starting set at the root of the binary tree consisting of all pivots, $P_1 \ldots P_n$. Let S_0, S_1 be the left and right children of S. Let *compset* be the ordered compatible set obtained directly from the incompatible table for the set S. Similarly, let $compset_0$ and $compset_1$ be the ordered compatible sets corresponding to S_0 and S_1 respectively.

A pivot P_j can split a set S iff:

$P_j \in S$ and
{set of incompatibles with P_j} $\cap$ S $\neq$ empty.

Assume P_j splits S into S_0 and S_1 ; then:

S_0 = S - {set of incompatibles with P_j } and
S_1 = S - [P_j].

There are two cases to consider:

i. P_j not in *compset*

The table corresponding to S_1 consists of the same null columns and compatible pivots as in *compset* so:

$$compset = compset_1.$$

ii. $P_j \in compset$

Then we must have:

$$imptbl_{P_j} \bigcap compset = \text{empty}$$

since P_j is compatible with all pivots in *compset* . In this case, $compset_0$ obtained from S_0 is equal to *compset* . We know P_j is in the set S_0 and that the incompatible table for S_0 is the same as the table for the parent set S with those rows and columns corresponding to incompatibles of P_j eliminated. Thus, all the compatible information which resulted in production of *compset* is transferred from the parent set S to S_0 and consequently:

$$compset = compset_0.$$

The above argument proves that, at level 1, one of the sets S_0 or S_1 will produce the same ordered compatible set as produced by its parent set. This proof holds for any two children of a set. In other words, at any point in the tree, an ordered compatible set corresponding to a parent set is reproduced by one of its children.

Induction on level verifies that generating the ordered compatible sets for every set from the root through level L of the tree does not produce any more information than producing the ordered compatible sets for every set at level L only.

As a consequence of the theorem, we generate all the sets at a given level in the binary tree, and for each set, we produce an ordered compatible set from the ordered incompatible table. Among the generated compatible sets we choose the set of largest size and lowest Markowitz sum to reduce the matrix and call the the resulting set the elimination set.

If we note that we split each set at each level of the tree for a given pivot according to its incompatibility information, then generation of the starting sets at different levels could be done in various ways:

i. We could split the starting sets using the original pivot ordering given by the input sparse matrix. This would generate completely random results.

ii. The same ordering used to order the incompatible table could be used to split the sets. This left to right ordering does not seem to agree with our low Markowitz sum requirement. At each split (level in the tree), we include one of the pivots, say p_j , with highest number of incompatibles (highest Markowitz number) in the left subtree. This inclusion also means we take a large number of pivots incompatible with p_j out of the sets in the left subtree. These pivots that are incompatible with p_j have lower Markowitz numbers than p_j and could themselves be compatible with some other elements in the set. As a result, this ordering will produce a left set considerably smaller in size than the resulting right set. Moreover, the left set contains pivots of high Markowitz number which would produce many fills if used to reduce the matrix. Therefore, some of the large compatible sets with small Markowitz sums cannot be generated from one of the sets in the left subtree unless we search very deep in the tree. In this case, the desired compatible sets would be in one of the right subtrees.

iii. A third alternative would be to split the sets with pivots in increasing order of their Markowitz numbers. Of course, in this case, the incompatibility information of the pivots used to split a starting set is taken from the right end of the incompatible table. Thus the complete incompatibility information for a pivot i is obtained by concatenating the row and column i of the table. This process, seems to give a better balance to the binary tree for the first few levels used to generate the starting sets required in our algorithm. Furthermore, it has the property that does not ignore pivots of low Markowitz numbers.

The high level description of this algorithm is given below:

Program Parallel Pivoting
- calculate Markowitz numbers of pivots in the remaining unreduced matrix.
- SORT pivots in decreasing order of Markowitz numbers
- produce all starting sets at level ULEVEL taking the pivots to split the sets from the root to ULEVEL in order of increasing Markowitz numbers.
- for each set at ULEVEL produce an ordered compatible set from the updated ordered incompatible table.
- among the ordered compatible sets generated above choose the maximum sized set with minimum Markowitz sum (Elimination set).

Here, ULEVEL is a preset level number indicating the depth of the tree to be searched. The algorithm is no longer exponential in time. An efficient implementation of the required sort and set operations are important factors in efficient execution of the algorithm. The set operations used in the construction of the incompatible table are of order 1 (adding an element to the set or a test for membership). The incompatible table can therefore be constructed in time nz, where nz is the number of nonzero elements of the matrix. Generation of an ordered compatible from the incompatible table requires scanning n sets corresponding to the columns of the table, and performing intersection and difference operations on the sets. These operations are of order n

with a constant factor equal to the inverse of the number of bits per computer word. The set operations are usually implemented in machine language or micro code and thus have a small time factor. They could be considered to have a constant time (rather than order of n) compared to the time taken to execute a high level language statement. Production of all starting sets for level ULEVEL takes a constant time. Generation of an ordered compatible for each starting set at ULEVEL takes a constant times n as explained above. For reasonable values of ULEVEL, all ordered compatible sets can be derived in parallel for different starting sets. In the next section we will see that good results are obtained for small, constant values of ULEVEL compared to n. The complexity of the algorithm is bounded above by the sorting algorithm. Thus, employing an efficient parallel sort would improve the performance of the new algorithm.

Balance between Parallelism and Fill-in Generation

Even though the above procedure tends to produce large sets of low Markowitz sums, we still could optimize the generation of fill-ins by considering a subset of the elimination set. That is, there could still be some room for trading off between parallelism and fill generation. To accomplish this task, we need to come up with parameters to control the number of pivots to be processed in parallel and the number of fills to be generated. One such parameter could be the size of the set of compatible pivots. By allowing a percentage of the set to be discarded, we can control the the number of compatible pivots to a degree that does not limit our parallel work by too much. For clarity, this parameter is called the shrinkage parameter and is used as a lower limit to shrink the elimination set by a percentage of its size. A different parameter could be an upper limit on the size of the elimination set. This limit would allow just enough work to keep our parallel processes busy. Of course shrinking of the elimination set must not be done arbitrarily by throwing pivots out of the set. In general, we would like to shrink our set by discarding pivots that would cause generation of many fills. Such pivots tend to have high Markowitz numbers. We already have pivots ordered according to their Markowitz numbers. We could use this ordering to scan pivots with highest Markowitz number in the elimination set and test against a threshold value. If pivots with Markowitz numbers greater than a threshold exist and if our shrinkage parameter allows, they are discarded from the elimination set. Use of a threshold value will allow us not to shrink a set that consists of all good pivot candidates of reasonably low Markowitz numbers. To serve this purpose, the threshold value should be set in comparison with low and high Markowitz numbers of the pivoting elements in the matrix. Again the ordered Markowitz numbers of pivots can be used to set such a threshold value conveniently. One way is to specify a fraction of candidates to be discarded from the elimination set. Consequently, we set the threshold to the Markowitz number of a pivot in a specific position in the list of pivot elements of the unreduced matrix (ordered by decreasing Markowitz numbers). Any pivot above this point in the ordered list is considered to have a high Markowitz number and therefore is a candidate for being discarded from the set, and any pivot below this point is considered acceptable.

Pivots in the elimination set are scanned in order of their highest Markowitz number. If a pivot with Markowitz number greater than the threshold exists and if the set is not already of minimum size, it is discarded from the set. The process is repeated until either no more pivots of large Markowitz numbers are left in the set or the set cannot be further shrunk. In the next section we present the result of different strategies and various parameters discussed here for a number of test matrices.

Analysis of the Results

The complexity of the binary tree search algorithm to obtain maximal compatible sets was such that it could not be run to completion for a 38 by 38 matrix. To verify the validity of our heuristic program, we performed every analysis described in this section on the small test matrices of Table 1.1. Recall that the new algorithm produces a number of starting sets for a given level (ULEVEL) of the binary search tree. For each starting set, an ordered compatible set is produced. Among the generated ordered compatible sets, the set with maximum size and minimum Markowitz sum is selected as the elimination set at that parallel step. Two alternative orderings for generation of starting sets at ULEVEL were discussed earlier. For simplicity, we call the algorithm to reduce a sparse matrix by compatible pivots using the decreasing order of Markowitz numbers for starting set splitting, DCOMP. Similarly, the algorithm which uses the increasing order of Markowitz numbers is called ICOMP.

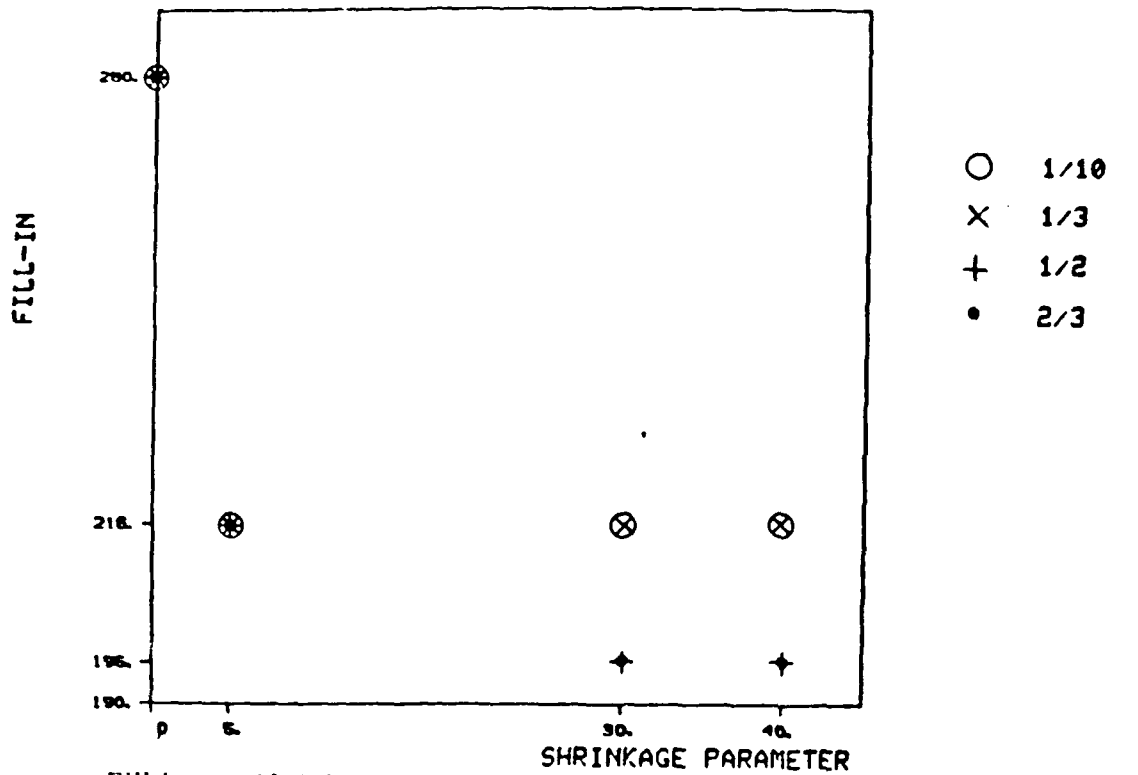
Detailed information produced by DCOMP and ICOMP are presented for three sparse matrices in Table 2.1. Column one of the table gives a description of the sparse matrix under consideration. Column 2, specifies the parallel step. Columns 3, 4, and 5 give the number of compatible pivots in the elimination set, its Markowitz sum and number of fill-ins generated at each step for program DCOMP. Similar information is summarized in the next three columns for program ICOMP. The information presented here is for ULEVEL=4. The first two matrices have been completely analyzed in the previous section and are presented here to show the validity of our proposed algorithms. It is interesting to see that, for the first matrix, DCOMP produced exactly the same results as the complete tree search program. On the other hand, ICOMP produced different results. Even though ICOMP produces a smaller compatible set in the first step, it finds larger sets in the next steps and reduces the same number of rows (i.e., 21) in five parallel steps. ICOMP generates 22 fills, almost half the number produced by DCOMP (40) or even the complete binary tree search algorithm (40). The same behavior is observed from the second 24 by 24 matrix. The third matrix is obtained from the circuit of an 8-bit full adder and is a 144 by 144 matrix with 616 nonzeros. Note that both algorithms produced an elimination set of 72 pivots in the first step, and so, half of the matrix can be reduced in parallel in one step. In this case the advantage of ICOMP over DCOMP is not significant.

To see how variation of depth will affect the resulting compatible sets, we ran both programs for values of ULEVEL between 2 and 5, for a number of matrices. These results are summarized in Table 2.2. Again the first column describes the matrix. The second column specifies ULEVEL.

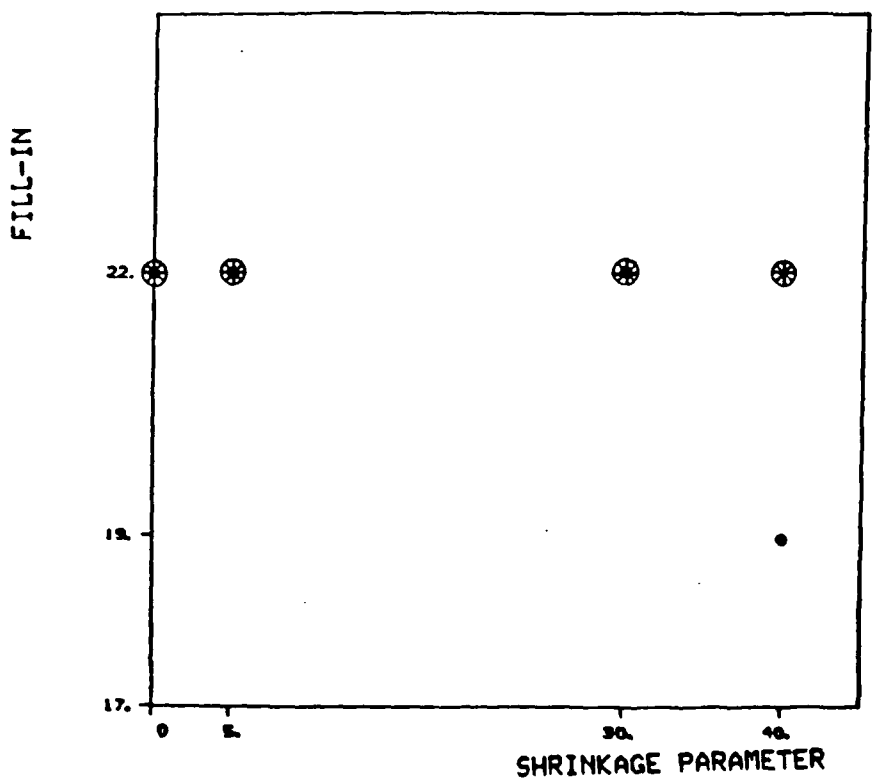
Columns 3 to 6 give related information for the DCOMP program. Here the first and the third columns specify number of parallel steps taken to reduce the matrix and number of rows reduced in those steps, respectively. The second column is the average parallel work at each step and is obtained by dividing total number of rows reduced in parallel by the number of parallel steps. The fourth column gives the total number of fill-ins generated by parallel reduction. The next four columns of the table provide similar information for the ICOMP program. The last two matrices of the table are produced from the SPAR program, which is a structural analysis program [12]. These two matrices have a peculiar block structure. Our initial objective was to study sparse matrices arising from SPICE. These matrices ordinarily have a random sparsity structure, but at the same time, the limited connectivity between nodes of the input circuit results in a limited number of nonzeros per row/column. The SPAR matrices will provide some insight into the behavior of our heuristic algorithms for a wider class of matrices.

It is clear from the table that, in almost every case, ICOMP produces better results both in terms of number of rows reduced in parallel and number of fill-ins generated. As was expected, DCOMP finds elimination sets of lower Markowitz sums as we search deeper in the tree. This is observed from the first 24 by 24 matrix and from the last SPAR generated 505 by 505 matrix. In the first matrix, ICOMP produced 18 fills, reducing 21 rows in 5 parallel steps, while DCOMP generated more than twice the number of fills and reduced 20 rows in 5 steps. The number of fills decreases for DCOMP as ULEVEL is increased, while ICOMP takes the opposite direction. This also shows that reasonably acceptable compatible sets, both in terms of size and Markowitz sum, are generated for small values of ULEVEL and it is not necessary to search very deep in the tree. The above observations hold for every matrix presented in the table, except the 78 by 78 matrix produced by the SPAR program. This matrix does not have characteristics typical of SPICE generated matrices; but, as we will see in our next analysis, acceptable results are produced for this matrix as well. Note that there are cases for the DCOMP program in which a higher average parallelism is indicated in the table than for ICOMP. In those situations, it is often the case that fewer rows have been reduced by DCOMP than by ICOMP.

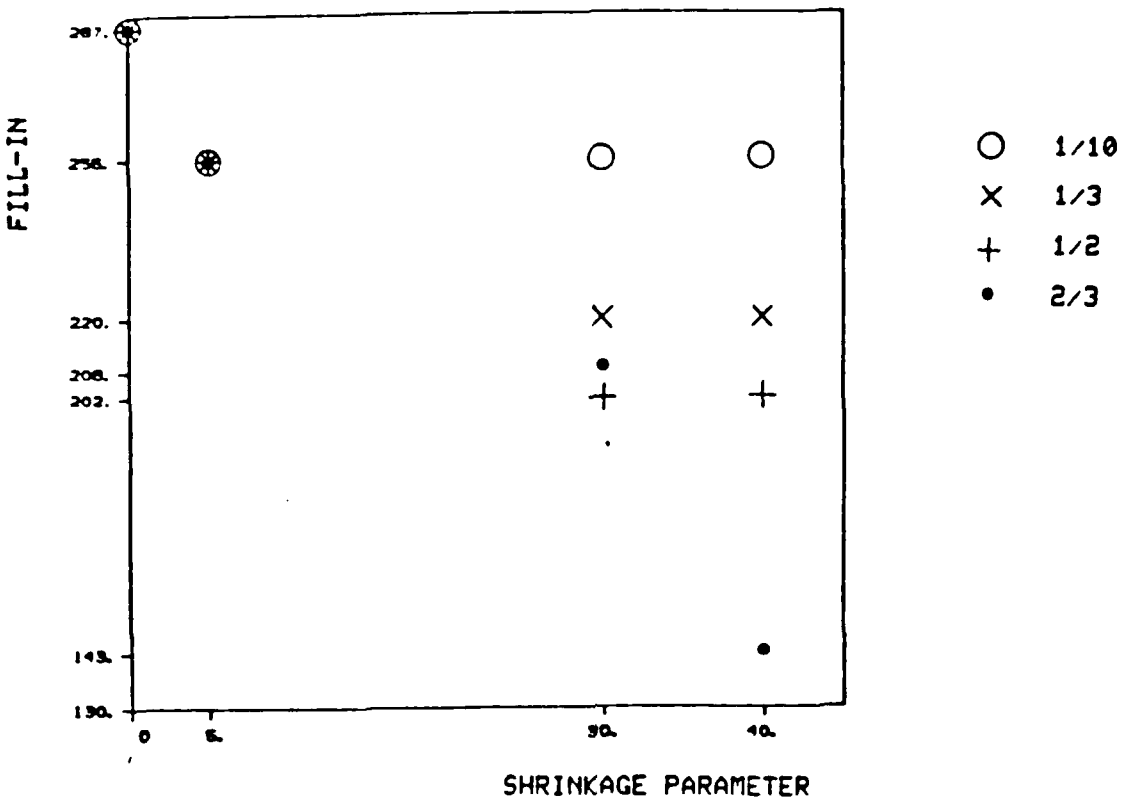
The remaining analyses are performed on the ICOMP program only, since it produces better results. In what follows, a value of 4 is used for ULEVEL. The next step is to study the effects of varying the parameters proposed earlier to obtain a balance between generation of fill-ins and the amount of parallel work. Results are summarized in Figures 2.3 to 2.6 for four of the matrices of Table 2.2. In these graphs, four different symbols are used to represent four different values of the threshold parameter. Recall that the threshold is set to the Markowitz number of a specific pivot in the ordered list of pivot candidates. On the graphs, the threshold value is given as a fraction of the pivoting elements in the remaining unreduced matrix, ordered in order of decreasing Markowitz numbers. For example when the threshold is 1/3, the Markowitz number of the pivot residing in the 1/3 point of the ordered list of pivot candidates in the unreduced matrix is obtained. Any pivot in the elimination set with Markowitz number greater than this



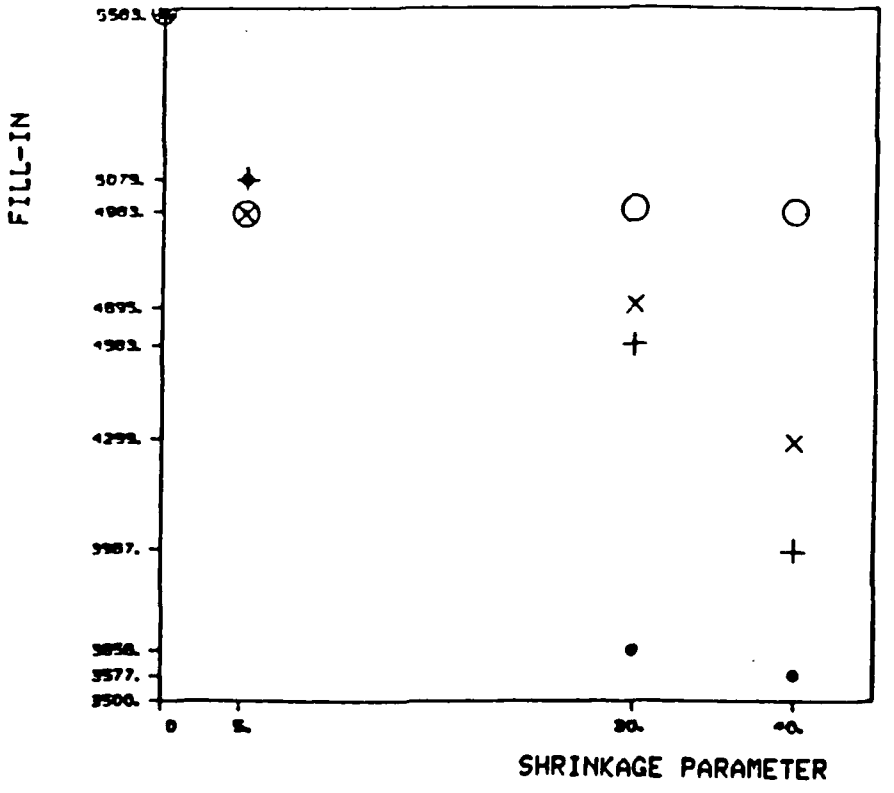
Fill-ins vs. Shrinkage for Different Thresholds,ULEVEL=4
144 by 141 Matrix of 8-bit Full Adder
Fig. 2.3



Fill-ins vs. Shrinkage for Different Thresholds,ULEVEL=4
24 by 24 Matrix of MOS Amplifier AC/DC
Fig. 2.4



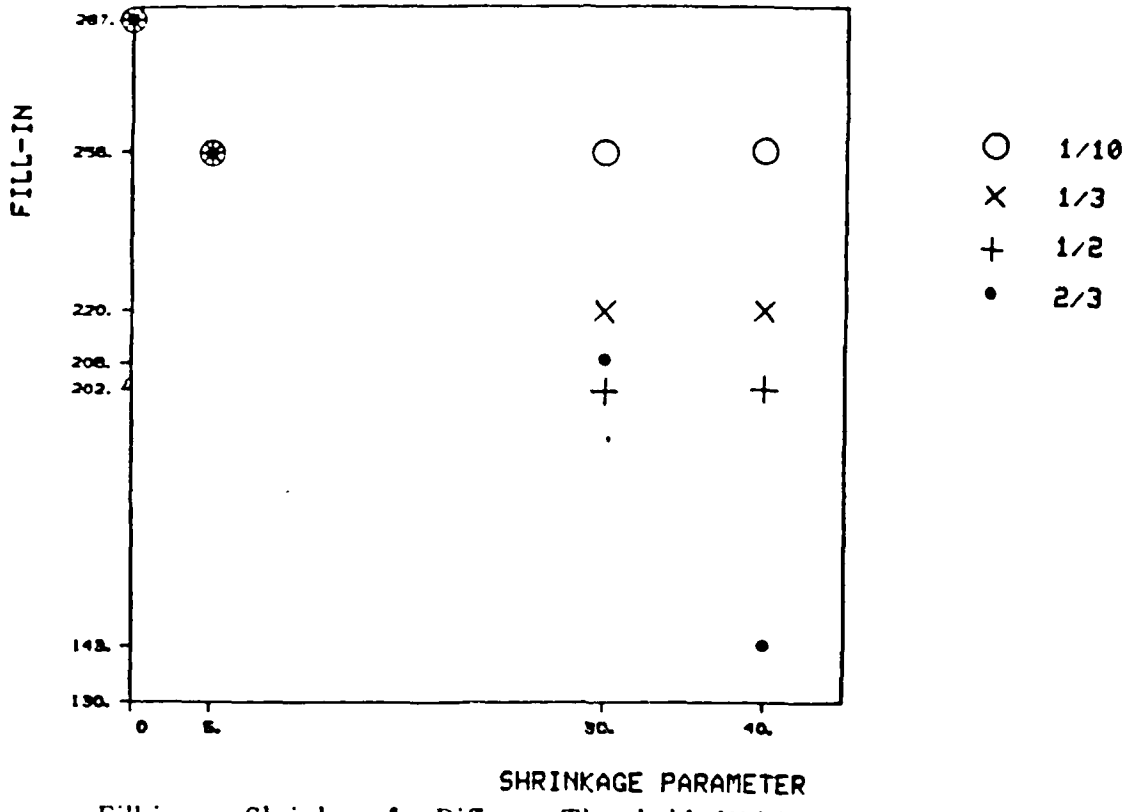
Fill-ins vs. Shrinkage for Different Thresholds,ULEVEL=4
78 by 78 Matrix of SPAR Program
Fig. 2.5



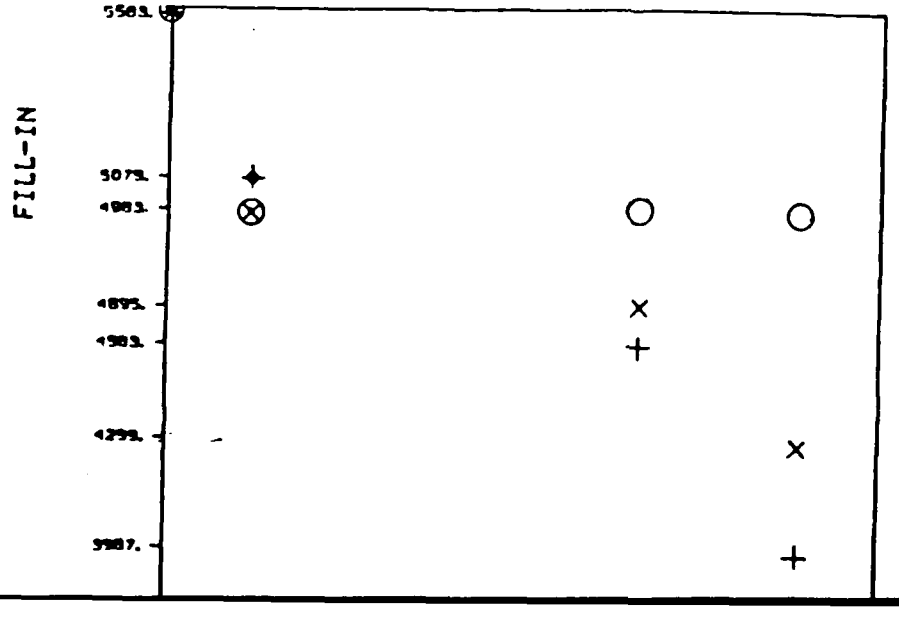
Fill-ins vs. Shrinkage for Different Thresholds,ULEVEL=4
505 by 505 Matrix of SPAR Program
Fig. 2.6

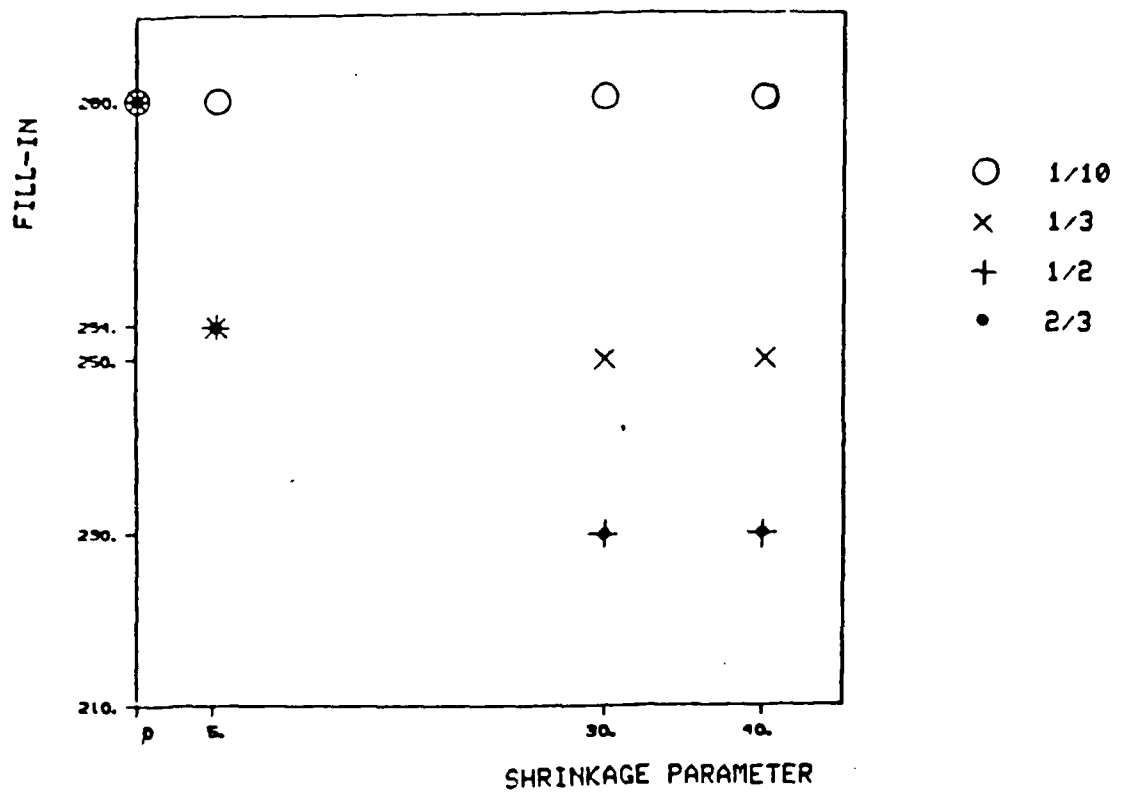
value is a candidate to be discarded from the set.

The graphs present information about the number of generated fill-ins versus the shrinkage parameter. In each case, the analysis is performed for threshold values of 1/10, 1/3, 1/2, and 2/3. For every value of the threshold,

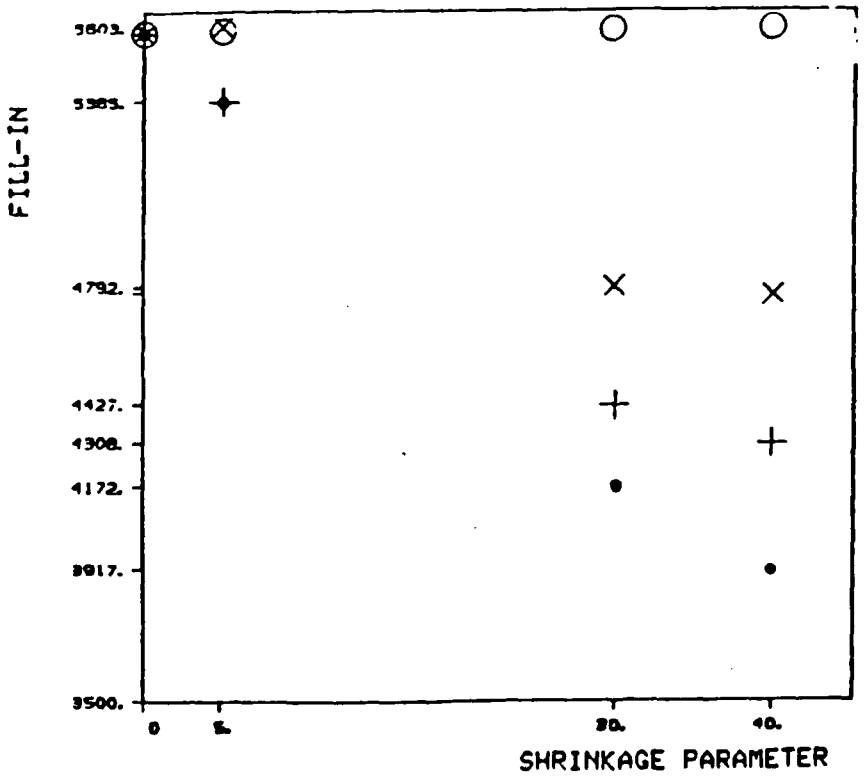


Fill-ins vs. Shrinkage for Different Thresholds, ULEVEL = 4
78 by 78 Matrix of SPAR Program
Fig. 2.5





Fill-ins vs. Shrinkage for Different Thresholds, ULEVEL = 4
no upper Limit, 144 by 144 Matrix of 8-bit Full Adder
Fig. 2.7



Fill-ins vs. Shrinkage for Different Thresholds, ULEVEL = 4
no Upper Limit, 505 by 505 Matrix of SPAR Program
Fig. 2.8

compatible sets consist of pivoting elements of lower Markowitz numbers than the highest one tenth. These graphs show more variation with the threshold parameter, even though a higher number of fill-ins is produced. Without exception, the fewest fill-ins are produced for the largest values of the threshold and the shrinkage parameter.

The number of fill-ins produced by ICOMP compares reasonably with sequential runs on the same matrices. For example, the sequential run on the 8-bit full adder matrix produced 166 fills, and ICOMP produced 196, which is an increase of about 18%. In general, as expected, the number of fill-ins produced by ICOMP is higher than the sequential results, but the difference is not great.

Conclusion

Solution of sparse systems of equations is essential in many application programs. Often such a system has to be solved repeatedly. In this paper we verified that in sparse matrices arising from electronic circuits it is possible to do computations on many diagonal elements simultaneously. A complete analysis of some test matrices, done by generating all maximal compatible sets of pivot elements, indicated the existence of many compatible pivots in these matrices. We have shown our test matrices do not become full during the decomposition. Furthermore, it was shown that many parallel computation steps are possible, and during these steps, the matrix is often reduced completely. The competing issues of parallel pivoting and fill-in generation have been studied, and we verified through examples that it is possible to reduce the production of fill-ins by removing some of the parallel pivot candidates from the elimination set on the basis of high Markowitz numbers. A heuristic algorithm was then proposed to produce large compatible sets of low Markowitz sums by a combination of an ordered partial tree search strategy and generation of ordered compatible sets. Different orderings to produce the ordered compatible sets were suggested, and their advantages and disadvantages were discussed and verified through the simulated results. A number of parameters to provide a balance between generation of fill-ins and the amount of parallel work were suggested, and their effects were determined in the simulated results.

The incompatible table required by the algorithm can be constructed in time nz (number of nonzero elements of the matrix). Production of starting sets for a given ULEVEL takes a constant time. For ULEVEL small and constant compared to n, generation of ordered compatibles from starting sets is of order n set intersection and difference operations. Assuming efficient implementation of the set operations is available, the heuristic algorithm has a complexity bounded above by the sorting algorithm required in the program. Thus, employing an efficient parallel sort program would improve the total performance of the new algorithm. Nevertheless, our results show that many compatible pivots are produced for parallel reduction of the sparse matrices, and the process can be repeated until the matrix is almost completely reduced. In cases where the matrix is not completely reduced, the remaining submatrix is of such a small size that parallel operations have little effect. Significant reduction in generation of fill-ins is obtained by varying

the proposed parameters. Moreover, as the result of these parameters, a better balance between the number of compatible pivots generated at different steps was achieved, while the reduction in parallel work proved to be insignificant.

REFERENCES

[1] Duff, I. S., "MA28: A Set of Fortran Subroutines for Sparse Linear Equations," *AERE Report R.8730*, London (1977).

[2] Jordan, H. F., "Parallelizing a Sparse Matrix Package," *Report CSDG 81-1*, Computer System Design Group, Electrical and Computer Engineering Department, University of Colorado, Boulder (1981).

[3] Alaghband, G. and Jordan, H. F., "Parallelizing a Sparse Matrix Package," *Report CSDG 83-3*, Computer System Design Group, Electrical and Computer Engineering Department, University of Colorado, Boulder (June 1983).

[4] Calahan, D. A., "Parallel Solution of Sparse Simultaneous Linear Equations," *Proc. 11th Annual Allerton Conf. Circuits and System Theory*, pp. 729-735 (Oct. 1973).

[5] Huang, J. W. and Wing, O., "Optimal Parallel Triangulation of a Sparse Matrix," *IEEE Trans. on Circuits and Systems*, v. CAS-26, no. 9, pp. 726-732 (Sept. 1976).

[6] Wing, O. and Huang, J. W., "A Computation Model of Parallel Solution of Linear Equations," *IEEE Trans. on Computers*, v. C-29, no. 9, pp. 632-638 (July 1980).

[7] Jess, J. A. G. and Kees, H. G. M., "A Data Structure for Parallel L/U Decomposition," *IEEE Trans. on Computers*, v. C-31, no. 3, pp. 231-239 (March 1982).

[8] Peters, F.J., "Parallel Pivoting Algorithm for Sparse Symmetric Matrices," *Parallel Computing*, v.1, North-Holland, pp. 99-110 (1984).

[9] Nagel, L. W., "SPICE2: A Computer Program to Simulate Semiconductor Circuits," *Memorandum ERL-M520*, Electronics Research Laboratory, University of Calif., Berkeley, CA 94720 (1975).

[10] Lewin, D., *Logical Design of Switching Circuits*, American Elsevier Publishing, New York, (1974).

[11] Markowitz, H. M., "The Elimination Form of the Inverse and It's Application to Linear Programming," *Management Sci.*, v. 3, pp. 255-269 (April 1957).

[12] "SPAR," *NASA CR 158970-1*, Engineering Information Systems Inc., San Jose, CA (Dec. 1978).

APPENDIX A

Parallel Pivoting Strategies										
Matrix	Step	Markowitz Sum			Ored Markowitz			No.of Sets	SPICE	
		NO.of Pivots	OP Count	NO.of Fill-in	NO.of Pivots	OP Count	NO.of Fill-in		Fill-in	% Sparse
Differential Pair	1	6	23	6	6	48	8	59		
	2	4	4	0	4	15	4	14		
16 by 16 Nz 57	3	3	5	0	2	0	0	4		
	4	2	2	0	2	2	0	1		
total	4	15		6	14		12		0	80.27
Cascaded, BTL Inverter	1	5	12	6	5	24	6	24		
	2	4	8	4	4	9	4	4		
12 by 12 Nz 34	3	2	2	0	2	1	0	1		
total	3	11		10	11		10		0	79.88
ECL Compatible SCHMITT Trigger	1	8	39	16	8	80	16	122		
	2	4	18	4	4	30	10	16		
	3	2	8	0	2	16	4	6		
18 by 18 Nz 66	4	2	5	0				2		
total	4	16		20	14		30		6	80.05

TABLE 1.1

Parallel Pivoting Strategies										
Matrix	Step	Markowitz Sum			Ored Markowitz			No.of Sets	SPICE	
		NO.of Pivots	OP Count	NO.of Fill-in	NO.of Pivots	OP Count	NO.of Fill-in		Fill-in	% Sparse
MOS Memory Cell	1	4	0	0	4	0	0	20		
	2	4	0	0	4	0	0	6		
13 by 13 Nz 47	3	3	6	2	3	4	2	3		
total	3	11		2	11		2		0	76.02
MOS Amplifier, AC/DC	1	8	169	36	8	156	36	173		
	2	5	117	4	5	61	4	12		
	3	3	0	0	3	0	0	5		
24 by 24 Nz 154	4	3	0	0	3	0	0	4		
	5	2	13	0	2	9	0	3		
total	5	21		40	21		40		10	73.76
MOS Amplifier Transient	1	8	178	40	8	156	40	149		
	2	4	91	4	4	72	4	14		
	3	3	0	0	3	0	0	5		
21 by 21 Nz 158	4	3	0	0	3	0	0	4		
	5	2	25	2	2	16	4	3		
total	5	20		46	20		48		22	71.20

TABLE 1.1
Continued

Fill-in Statistics				
Matrix	Step	Set size	OP.Count	Fill-in
16 by 16 Nz 57	1	6 5 4	23 14 5	6 2 0
	2	4	4	0
	3 4	3 2	5 2	0 0
12 by 12 Nz 34	1	5 4 3 2	12 8 4 0	6 4 2 0
	2	4 3 2	8 4 0	4 2 0
	3	2	2	0
18 by 18 Nz 66	1	8 7 6 5 4	39 30 21 12 8	16 10 6 2 0
	2	4 3 2	18 9 0	4 2 0
	3	2	8	0
	4	2	5	0

TABLE 1.2

Fill-in Statistics				
Matrix	Step	Set size	OP.Count	Fill-in
13 by 13 Nz 47	1	4	0	0
	2	4	0	0
	3	3	6	2
		2	2	0
24 by 24 Nz 154	1	8	169	36
		7	120	24
		6	84	18
		5	59	10
		4	34	6
		3	18	2
		2	9	0
	2	5	117	4
		4	68	0
	3	3	0	0
	4	3	0	0
	5	2	13	0
24 by 24 Nz 158	1	8	178	40
		7	129	28
		6	93	20
		5	68	14
		4	43	8
		3	18	4
		2	9	2
	2	4	91	4
		3	55	4
		2	3	0
	3	3	0	0
	4	3	0	0
	5	2	25	2

TABLE 1.2
Continued

PARALLELISM vs. FILL-IN								
Matrix	Step	Max Size	Reduced Size	MAX OP Count	Reduced OP.Count	Fill-in	Mrkowitz Sum	SPICE
16 by 16 Nz 57	1 2 3 4	6 5 4 2	5 1 3 2	23 13 6	14 4 2	2 0 0 0		
total	4		14			2	6	0
12 by 12 Nz 34	1 2 3 4	5 4 3 2	4 3 2 2	12 8 6	8 4 2	4 2 0 0		
total	4		11			6	10	0
18 by 18 Nz 66	1 2 3 4	8 6 3 2	7 5 2 2	39 26 11	30 17 5	10 6 0 0		
total	4		16			16	20	6

TABLE 1.3

PARALLELISM vs. FILL-IN								
Matrix	Step	Max Size	Reduced Size	MAX OP Count	Reduced OP.Count	Fill-in	Markowitz Sum	SPICE
13 by 13 Nz 47	1	4	4	0	0	0		
	2	4	4	0	0	0		
	3	3	2	6	2	0		
	4	2	2			0		
total	4		12			0	2	0
21 by 21 Nz 151	1	8	7	169	120	24		
	2	5	4	93	57	2		
	3	3	3	0	0	0		
	4	3	3	0	0	0		
	5	2	2			0		
	6	2	2			0		
total	6		21			26	40	10
21 by 21 Nz 158	1	8	7	178	129	28		
	2	5	4	110	91	4		
	3	3	3	0	0	0		
	4	3	3	0	0	0		
	5	3	2	22	13	4		
	6	2	2			0		
total	6		21			36	16	22

TABLE 1.3
Continued

Matrix	Step	DCOMP			ICOMP		
		No.of Pivots	OP. Count	No.of Fill-in	No.of Pivots	OP. Count	No.of Fill-in
MOS	1	8	109	36	4	0	0
Amplifier	2	5	117	4	7	54	18
AC/DC	3	3	0	0	5	54	4
	4	3	0	0	3	0	0
24 by 24	5	2	13	0	2	10	0
Nz 154							
Total	5	21		40	21		22
MOS	1	8	178	40	4	0	0
Amplifier	2	4	119	10	7	61	17
Transient	3	3	0	0	5	60	8
	4	3	0	0	3	0	0
24 by 24	5	2	25	0	2	13	1
Nz 158							
Total	5	20		50	21		26
8-Bit	1	72	449	150	72	449	150
Full Adder	2	25	258	86	25	247	76
	3	16	99	0	16	99	0
	4	11	110	12	11	110	20
111 by 111	5	6	75	22	6	75	22
Nz 616	6	5	46	8	5	46	8
	7	3	29	4	3	29	4
	8	2	10	0	2	10	0
	9	2	8	0	2	8	0
Total	9	142		282	142		280

Comparison of the Two Proposed Orderings
for ULEVEL=4.

TABLE 2.1

Matrix	ULEVEL	DCOMP				ICOMP			
		Steps	Avg. Par.	Rows	No.of Fill-in	Steps	Avg. Par	Rows	No.of Fill-in
21 by 21 Nz 154	2	5	4	20	44	5	4.2	21	18
	3	5	4	20	44	5	4.2	21	22
	4	5	4.2	21	40	5	4.2	21	22
	5	5	4.2	21	40	6	3.6	22	20
21 by 21 Nz 158	2	5	4	20	50	5	4	20	20
	3	5	4	20	50	5	4.2	21	20
	4	5	4	20	50	5	4.2	21	20
	5	5	4	20	50	5	4.2	21	20
52 by 52 Nz 166	2	5	9	45	128	6	7.8	47	121
	3	5	9	45	128	6	7.8	47	121
	4	5	9	45	128	6	7.7	46	137
	5	6	7.8	47	123	6	7.7	46	137
141 by 141 Nz 616	2	9	15.8	142	282	9	15.8	142	258
	3	9	15.8	142	282	8	17.6	141	264
	4	9	15.8	142	282	9	15.8	142	280
	5	9	15.8	142	282	9	15.8	142	280
78 by 78 Nz 398	2	10	7.7	77	198	8	9.3	74	238
	3	8	9.3	74	200	10	7.5	75	271
	4	9	8.4	76	199	9	8.3	75	287
	5	10	7.7	77	202	9	8.3	75	292
505 by 505 Nz 5889	2	36	13.6	488	5802	40	12.3	493	5132
	3	37	13.2	490	5811	41	12.1	497	5561
	4	34	14.3	485	5802	41	12	489	5583
	5	38	13	495	5785	36	13.4	481	5695

Comparison of DCOMP and ICOMP for Different levels

TABLE 2.2

1 Report No. NASA CR-178016 ICASE Report No. 85-48	2. Government Accession No.	3. Recipient's Catalog No.
4 Title and Subtitle Multiprocessor Sparse L/U Decomposition with Controlled Fill-In		5. Report Date October 1985 6. Performing Organization Code
7 Author(s) Gita Alaghband and Harry F. Jordan		8. Performing Organization Report No. 85-48 10. Work Unit No.
9 Performing Organization Name and Address Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23665-5225		11. Contract or Grant No. NAS1-17070, AFOSR 85-0189 13. Type of Report and Period Covered Contractor Report
12 Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, D.C. 20546		14. Sponsoring Agency Code 505-31-83-01

15 Supplementary Notes
Langley Technical Monitor: J. C. South Jr.
Final Report
Submitted to IEEE Trans. on Computers

16 Abstract

During L/U decomposition of a sparse matrix, it is possible to perform computation on many diagonal elements simultaneously. Pivots that can be processed in parallel are related by a compatibility relation and are grouped in a compatible set. The collection of all maximal compatibles yields different maximum sized sets of pivots that can be processed in parallel. Generation of the maximal compatibles is based on the information obtained from an incompatible table. This table provides information about pairs of incompatible pivots. In this paper, generation of the maximal compatibles of pivot elements for a class of small sparse matrices is studied first. The algorithm involves a binary tree search and has a complexity exponential in the order of the matrix. Different strategies for selection of a set of compatible pivots based on the Markowitz criterion are investigated. The competing issues of parallelism and fill-in generation are studied and results are provided. A technique for obtaining an ordered compatible set directly from the ordered incompatible table is given. This technique generates a set of compatible pivots with the property of generating few fills. A new hueristic algorithm is then proposed that combines the idea of an ordered compatible set with a limited binary tree search to generate several sets of compatible pivots in linear time. Finally, an elimination set to reduce the matrix is selected. Parameters are suggested to obtain a balance between parallelism and fill-ins. Results of applying the proposed algorithms on several large application matrices are presented and analyzed.

17 Key Words (Suggested by Author(s))	18. Distribution Statement
multiprocessor Gaussian elimination parallel pivoting	61 - Computer Programming & Software 64 - Numerical Analysis Unclassified - Unlimited

19 Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	38	A03

END

FILMED

9-89

DTIC